深入浅出Flink(1)

一、课前准备

1. 下载Flink版本 https://archive.apache.org/dist/flink/flink-1.12.0/flink-1.12.0-bin-scala_2.11.tgz

二、课堂主题

讲解Flink编程模型

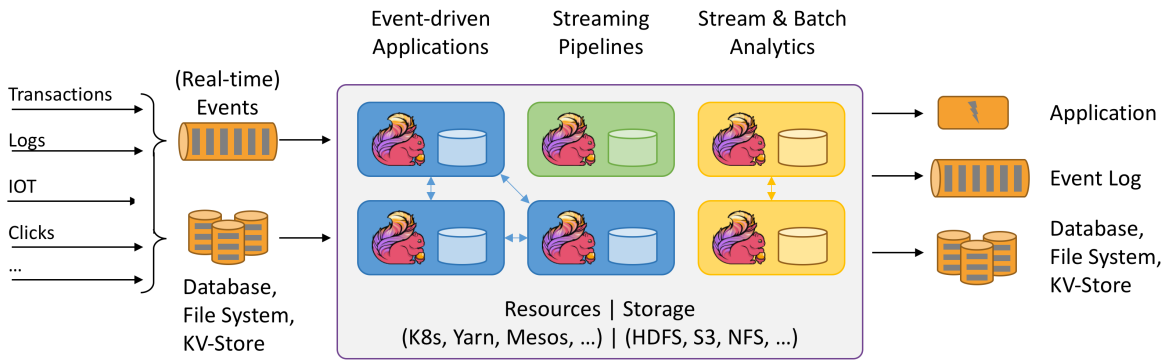
三、课程目标

1. 掌握Flink编程模型
2. 掌握Flink数据传输方式
3. 掌握Flink并行度原理

四、 知识要点

4.1 Flink简介

Apache Flink® — Stateful Computations over Data Streams



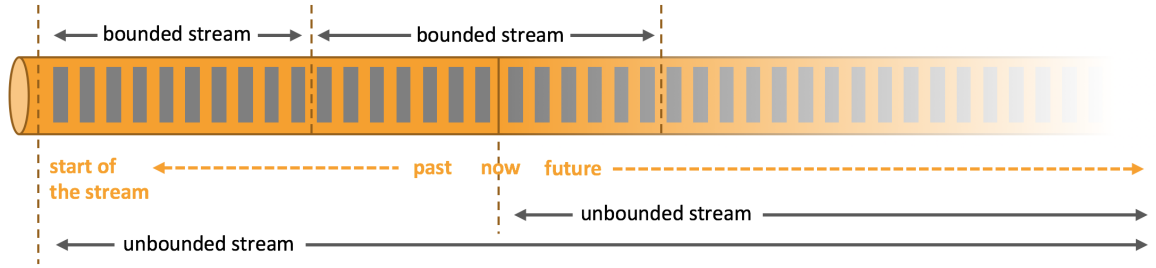
Apache Flink 是一个框架和分布式处理引擎，用于在*无边界和有边界*数据流上进行有状态的计算。Flink能在所有常见集群环境中运行，并能以内存速度和任意规模进行计算。

4.1.1 处理无界和有界数据

任何类型的数据都可以形成一种事件流。信用卡交易、传感器测量、机器日志、网站或移动应用程序上的用户交互记录，所有这些数据都形成一种流。

数据可以被作为 *无界* 或者 *有界* 流来处理。

1. **无界流**(实时流，实时的程序) 有定义流的开始，但没有定义流的结束。它们会无休止地产生数据。无界流的数据必须持续处理，即数据被摄取后需要立刻处理。我们不能等到所有数据都到达再处理，因为输入是无限的，在任何时候输入都不会完成。处理无界数据通常要求以特定顺序摄取事件，例如事件发生的顺序，以便能够推断结果的完整性。
2. **有界流** 有定义流的开始，也有定义流的结束。有界流可以在摄取所有数据后再进行计算。有界流所有数据可以被排序，所以并不需要有序摄取。有界流处理通常被称为批处理



Apache Flink 擅长处理无界和有界数据集 精确的时间控制和状态化使得 Flink 的运行时(runtime)能够运行任何处理无界流的应用。有界流则由一些专为固定大小数据集特殊设计的算法和数据结构进行内部处理，产生了出色的性能。

4.1.2 部署应用到任意地方

Apache Flink 是一个分布式系统，它需要计算资源来执行应用程序。Flink 集成了所有常见的集群资源管理器，例如 Hadoop YARN、 Apache Mesos 和 Kubernetes，但同时也可以作为独立集群运行（standalone）。

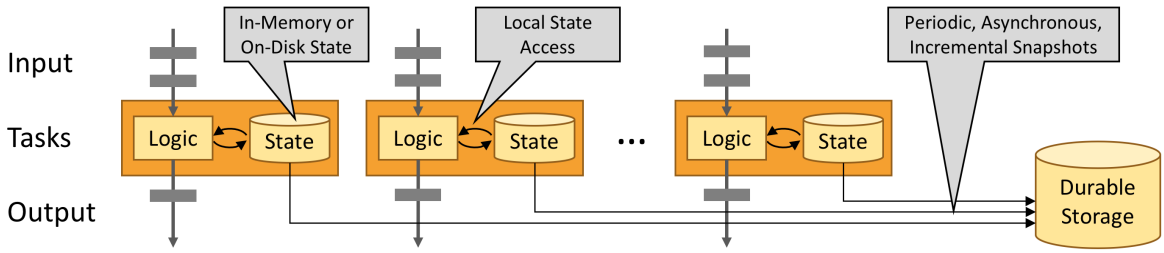
Flink 被设计为能够很好地工作在上述每个资源管理器中，这是通过资源管理器特定(resource-manager-specific)的部署模式实现的。Flink 可以采用与当前资源管理器相适应的方式进行交互。
部署 Flink 应用程序时，Flink 会根据应用程序配置的并行性自动标识所需的资源，并从资源管理器请求这些资源。在发生故障的情况下，Flink 通过请求新资源来替换发生故障的容器。提交或控制应用程序的所有通信都是通过 REST 调用进行的，这可以简化 Flink 与各种环境中的集成

4.1.3 运行任意规模应用

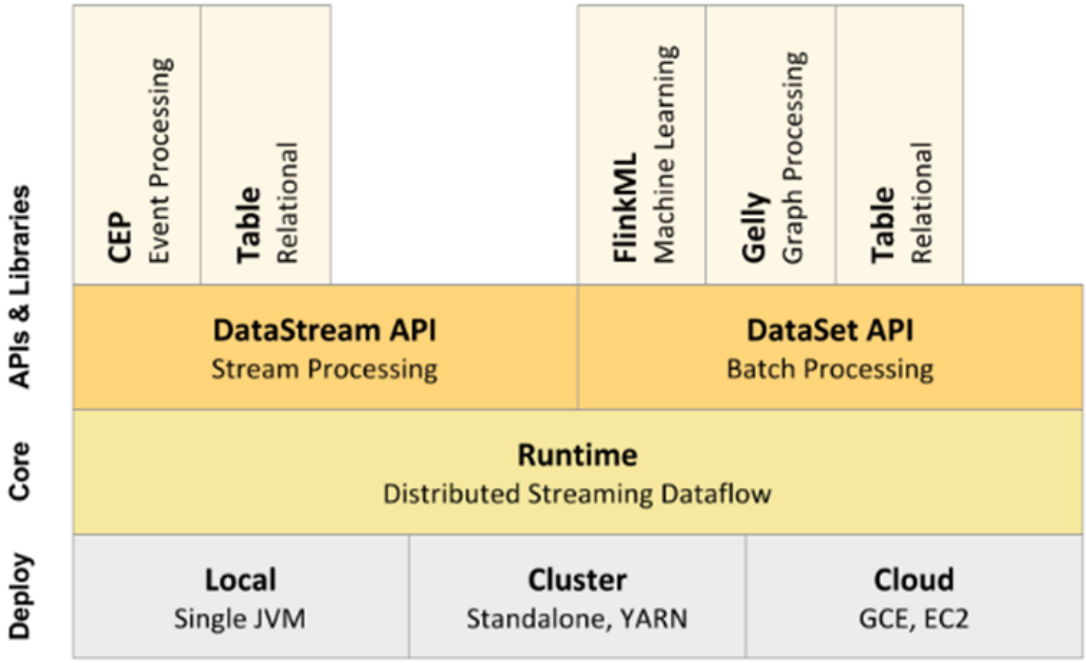
Flink 旨在任意规模上运行有状态流式应用。因此，应用程序被并行化为可能数千个任务，这些任务分布在集群中并发执行。所以应用程序能够充分利用无尽的 CPU、内存、磁盘和网络 IO。而且 Flink 很容易维护非常大的应用程序状态。其异步和增量的检查点算法对处理延迟产生最小的影响，同时保证精确一次状态的一致性。
Flink 用户报告了其生产环境中一些令人印象深刻的扩展性数字：
每天处理数万亿的事件
可以维护几TB大小的状态
可以部署上千个节点的集群

4.1.4 利用内存性能

有状态的 Flink 程序针对本地状态访问进行了优化。任务的状态始终保留在内存中，如果状态大小超过可用内存，则会保存在能高效访问的磁盘数据结构中。任务通过访问本地（通常在内存中）状态来进行所有的计算，从而产生非常低的处理延迟。Flink 通过定期和异步地对本地状态进行持久化存储来保证故障场景下精确一次的状态一致性。



4.2 Flink架构图



4.3 入门案例演示

4.3.1 pom文件

1. 官网建议使用IDEA，集成Scala和Maven比较方便
2. pom.xml文件指定

```
<properties>
        <flink.version>1.12.0</flink.version>
        <scala.version>2.11.8</scala.version>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

   <dependencies>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-streaming-java_2.11</artifactId>
            <version>${flink.version}</version>
        </dependency>
     </dependencies>

    <build>
        <pluginManagement>
            <plugins>
                <plugin>
                    <groupId>net.alchim31.maven</groupId>
                    <artifactId>scala-maven-plugin</artifactId>
                    <version>3.2.2</version>
                </plugin>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <version>3.5.1</version>
```

```
            </plugin>
        </plugins>
    </pluginManagement>
    <plugins>
        <plugin>
            <groupId>net.alchim31.maven</groupId>
            <artifactId>scala-maven-plugin</artifactId>
            <executions>
                <execution>
                    <id>scala-compile-first</id>
                    <phase>process-resources</phase>
                    <goals>
                        <goal>add-source</goal>
                        <goal>compile</goal>
                    </goals>
                </execution>
                <execution>
                    <id>scala-test-compile</id>
                    <phase>process-test-resources</phase>
                    <goals>
                        <goal>testCompile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>

        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <executions>
                <execution>
                    <phase>compile</phase>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>



        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>2.4.3</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <filters>
                            <filter>
                                <artifact>*:*</artifact>
                                <excludes>
                                    <exclude>META-INF/*.SF</exclude>
                                    <exclude>META-INF/*.DSA</exclude>
```

```
                                    <exclude>META-INF/*.RSA</exclude>
                                </excludes>
                            </filter>
                        </filters>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

4.3.2 单词计数案例演示

```
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.util.Collector;

public class WordCount {
    public static void main(String[] args) throws Exception {
        //步骤一：获取执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        //步骤二：获取数据源
        DataStreamSource<String> dataStream =
env.socketTextStream("192.168.152.102", 9999);
        //步骤三：数据处理
        SingleOutputStreamOperator<Tuple2<String, Integer>> wordAndOne =
dataStream.flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {
            @Override
            public void flatMap(String line, Collector<Tuple2<String, Integer>>
collector) throws Exception {
                String[] fields = line.split(",");
                for (String word : fields) {
                    collector.collect(new Tuple2<>(word, 1));
                }
            }
        });

        SingleOutputStreamOperator<Tuple2<String, Integer>> wordCount =
wordAndOne.keyBy(0)
                .sum(1);
        //步骤四：数据输出
        wordCount.print();
        //步骤五：启动任务
        env.execute("word count ...");
    }
}
```

4.3.3使用面向对象

把数据看成对象，遇到字段较多的数据操作比较方便

```
public class WordCount {
    public static void main(String[] args) throws Exception {
        //步骤一：获取执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        //步骤二：获取数据源
        DataStreamSource<String> dataStream =
env.socketTextStream("192.168.152.102", 9999);
        //步骤三：数据处理
        SingleOutputStreamOperator<WordAndCount> wordCount =
dataStream.flatMap(new FlatMapFunction<String, WordAndCount>() {
            @Override
            public void flatMap(String line, Collector<WordAndCount> collector)
throws Exception {
                String[] fields = line.split(",");
                for (String word : fields) {
                    collector.collect(new WordAndCount(word, 1));
                }
            }
        }).keyBy("word")
                .sum("count");
        //步骤四：数据输出
        wordCount.print();

        //步骤五：启动任务
        env.execute("word count ...");
    }


    public static class WordAndCount{
        private String word;
        private int count;

        public WordAndCount(){

        }
        public WordAndCount(String word, int count) {
            this.word = word;
            this.count = count;
        }

        public String getWord() {
            return word;
        }

        public void setWord(String word) {
            this.word = word;
        }

        public int getCount() {
            return count;
        }

        public void setCount(int count) {
            this.count = count;
```

```
        }

        @Override
        public String toString() {
            return "WordAndCount{" +
                    "word='" + word + '\'' +
                    ", count=" + count +
                    '}';
        }
    }

}
```

4.3.4 使用最佳实践

flink建议如果程序中需要传入参数，使用它提供的ParameterTool

```
public class WordCount {
    public static void main(String[] args) throws Exception {
        //步骤一：获取执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        //步骤二：获取数据源
        //flink提供的工具类，获取传递的参数
        ParameterTool parameterTool = ParameterTool.fromArgs(args);
        String hostname = parameterTool.get("hostname");
        int port = parameterTool.getInt("port");
        DataStreamSource<String> dataStream =
env.socketTextStream(hostname,port);
        //步骤三：数据处理
        SingleOutputStreamOperator<WordAndCount> wordCount =
dataStream.flatMap(new FlatMapFunction<String, WordAndCount>() {
            @Override
            public void flatMap(String line, Collector<WordAndCount> collector)
throws Exception {
                String[] fields = line.split(",");
                for (String word : fields) {
                    collector.collect(new WordAndCount(word, 1));
                }
            }
        }).keyBy("word")
                .sum("count");
        //步骤四：数据输出
        wordCount.print();

        //步骤五：启动任务
        env.execute("word count ...");
    }


    public static class WordAndCount{
        private String word;
```

```
        private int count;

        public WordAndCount(){

        }
        public WordAndCount(String word, int count) {
            this.word = word;
            this.count = count;
        }

        public String getWord() {
            return word;
        }

        public void setWord(String word) {
            this.word = word;
        }

        public int getCount() {
            return count;
        }

        public void setCount(int count) {
            this.count = count;
        }

        @Override
        public String toString() {
            return "WordAndCount{" +
                    "word='" + word + '\'' +
                    ", count=" + count +
                    '}';
        }
    }

}
```

4.3.5 抽离业务功能

工作中开发复杂功能模块，习惯把业务算子抽离出来单独开发，这样代码结构会比较清晰

```
public class WordCount {
    public static void main(String[] args) throws Exception {
        //步骤一：获取执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        //步骤二：获取数据源
        //flink提供的工具类，获取传递的参数
        ParameterTool parameterTool = ParameterTool.fromArgs(args);
        String hostname = parameterTool.get("hostname");
        int port = parameterTool.getInt("port");
        DataStreamSource<String> dataStream =
env.socketTextStream(hostname,port);
```

```
        //步骤三：数据处理
        SingleOutputStreamOperator<WordAndCount> wordCount = dataStream
                .flatMap(new SplitWord())
                .keyBy("word")
                .sum("count");
        //步骤四：数据输出
        wordCount.print();

        //步骤五：启动任务
        env.execute("word count ...");
    }


    /**
     * 分割单词
     */
    public static class SplitWord implements
 FlatMapFunction<String,WordAndCount>{
        @Override
        public void flatMap(String line, Collector<WordAndCount> collector)
throws Exception {
            String[] fields = line.split(",");
            for (String word : fields) {
                collector.collect(new WordAndCount(word, 1));
            }
        }
    }



    public static class WordAndCount{
        private String word;
        private int count;

        public WordAndCount(){

        }
        public WordAndCount(String word, int count) {
            this.word = word;
            this.count = count;
        }

        public String getWord() {
            return word;
        }

        public void setWord(String word) {
            this.word = word;
        }

        public int getCount() {
            return count;
        }

        public void setCount(int count) {
            this.count = count;
        }
```

```
        @Override
        public String toString() {
            return "WordAndCount{" +
                    "word='" + word + '\'' +
                    ", count=" + count +
                    '}';
        }
    }

}
```

4.3.6 Scala版本演示

flink源码用的是Java写的，但是也支持Scala API，在企业中有些团队也会用Scala开发，所以我们授课过程中会两种语言都使用到，最后做到两种语言开发都能掌握。

pom.xml文件添加如下内容：

```
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-streaming-scala_2.11</artifactId>
            <version>${flink.version}</version>
        </dependency>
```

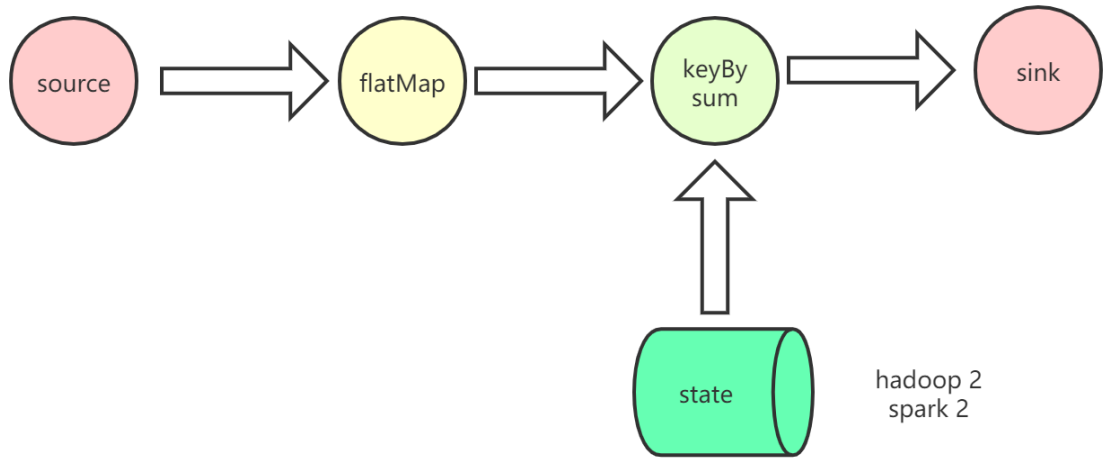
```
import org.apache.flink.api.java.utils.ParameterTool
import org.apache.flink.streaming.api.scala._

object WordCount {
  def main(args: Array[String]): Unit = {
    //获取参数
    val hostname = ParameterTool.fromArgs(args).get("hostname")
    val port = ParameterTool.fromArgs(args).getInt("port")
    //TODO 导入隐式转换
    //步骤一：获取执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //步骤二：获取数据源
    val textStream = env.socketTextStream(hostname,port)
    //步骤三：数据处理
    val wordCountStream = textStream.flatMap(line => line.split(","))
      .map((_, 1))
      .keyBy(0)
      .sum(1)
    //步骤四：数据结果处理
    wordCountStream.print()
    //步骤六：启动程序
    env.execute("WindowWordCountScala")
  }

}
```

4.4 Flink核心概念

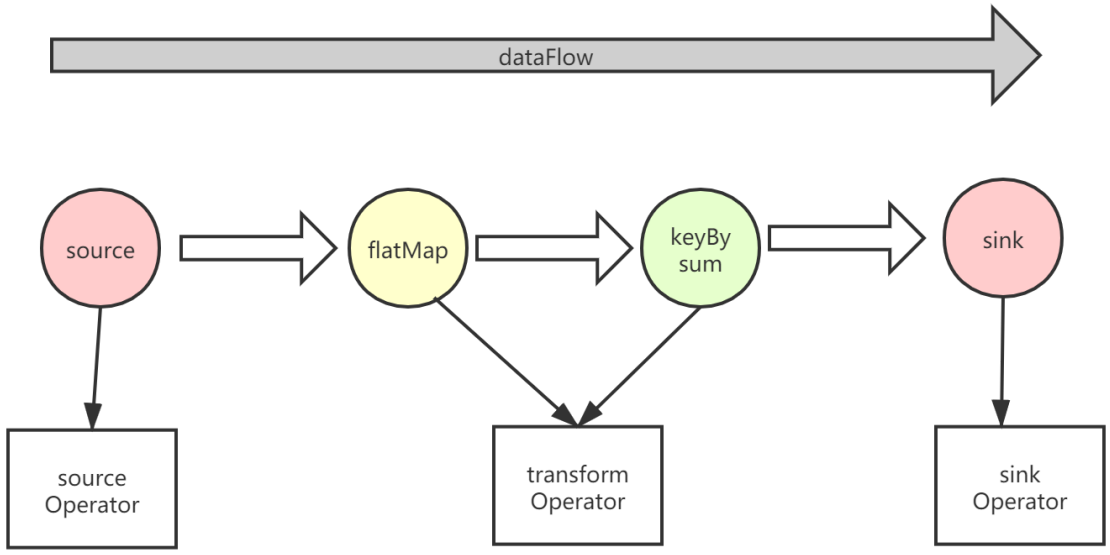
4.4.1 Flink概念

```
Apache Flink is a framework and distributed processing engine for stateful
computations over unbounded and bounded data streams
```

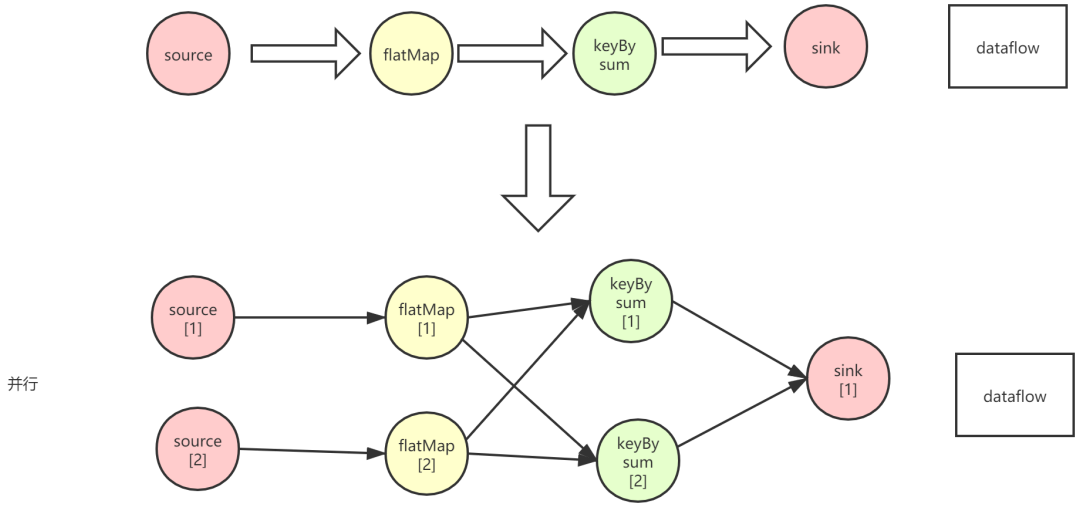
核心概念之stateful



核心概念之Operator



核心概念之distributed



4.4.2 本地观察Flink任务

步骤一：pom.xml中添加如下依赖

```
<dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-runtime-web_2.11</artifactId>
            <version>${flink.version}</version>
</dependency>
```

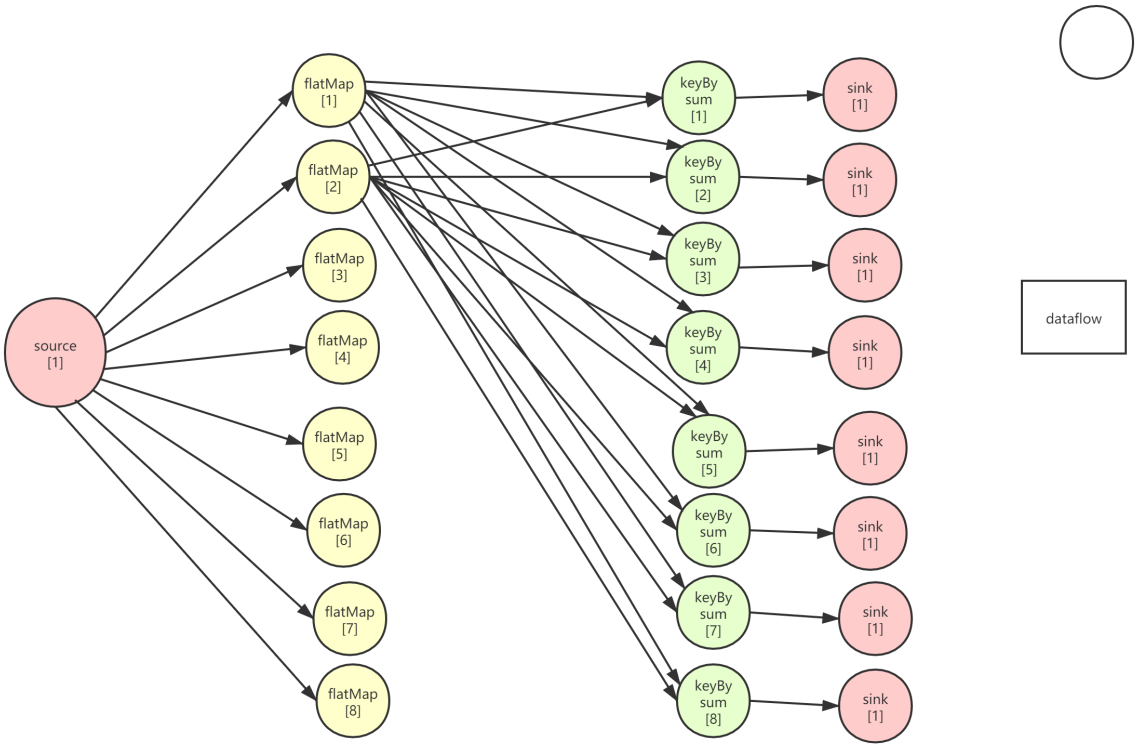
步骤二：修改获取执行环境的代码

代码修改为：

```
   StreamExecutionEnvironment env =
StreamExecutionEnvironment.createLocalEnvironmentWithWebUI(new Configuration());
```

步骤三：运行程序

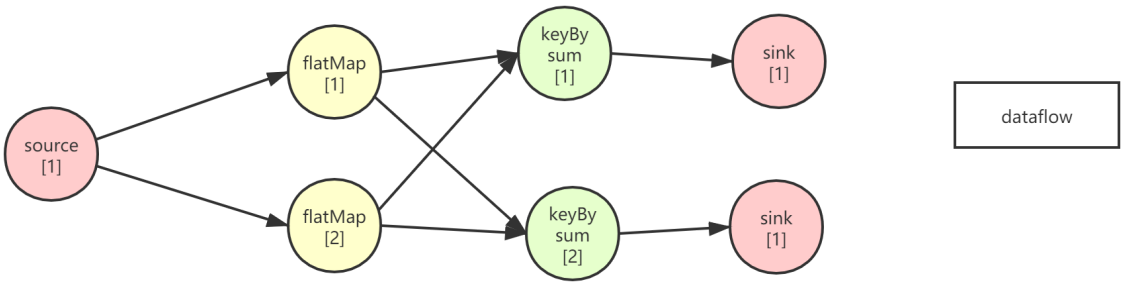
程序运行起来打开 http://localhost:8081/



4.4.3 并行度

1. 设置全局并行度为2

```
env.setParallelism(2);
```



2. 设置sink Operator并行度

```
wordCount.print().setParallelism(1);
```

3. Flink架构

Flink的架构是主从式的架构， 主节点叫：JobManager，从节点叫TaskManager

bIgdata02

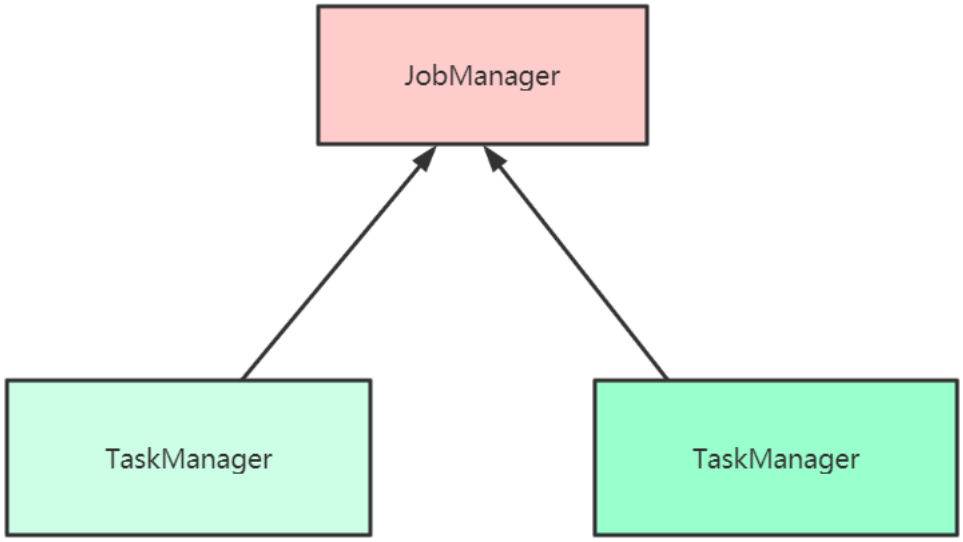
JobManager

bigdata03

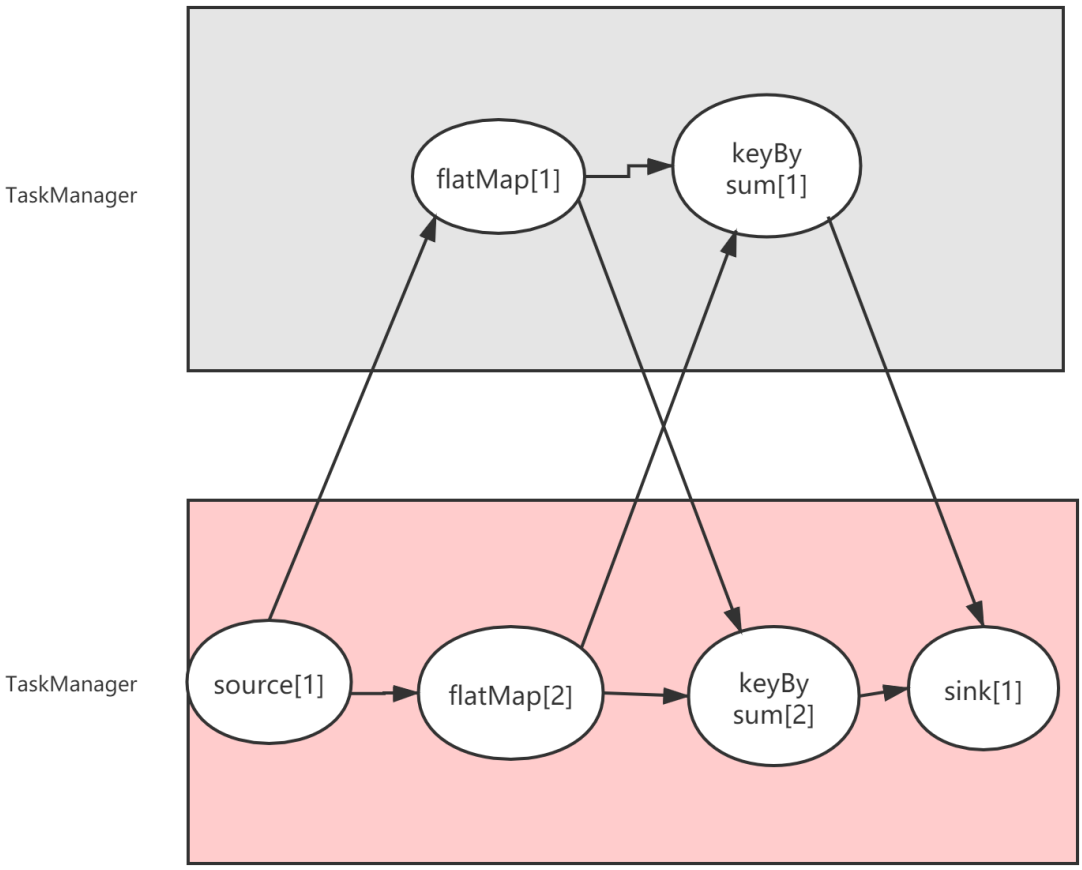
Taskmanager

bigdata04

TaskManager

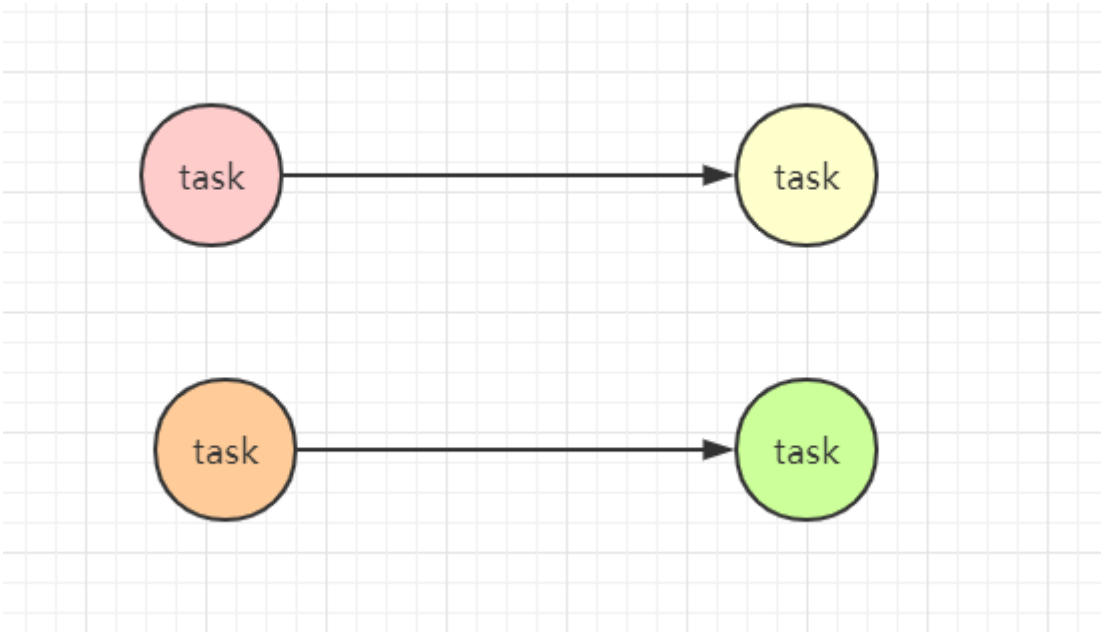


4. 任务分布式运行



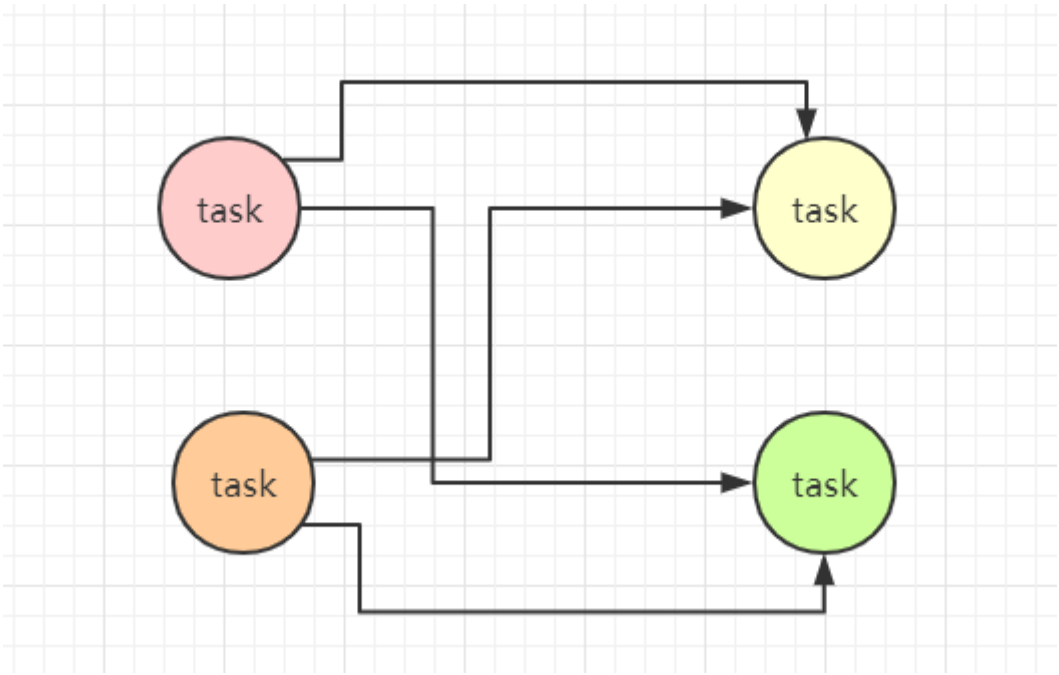
4.4.4 数据传输策略

1. forward strategy



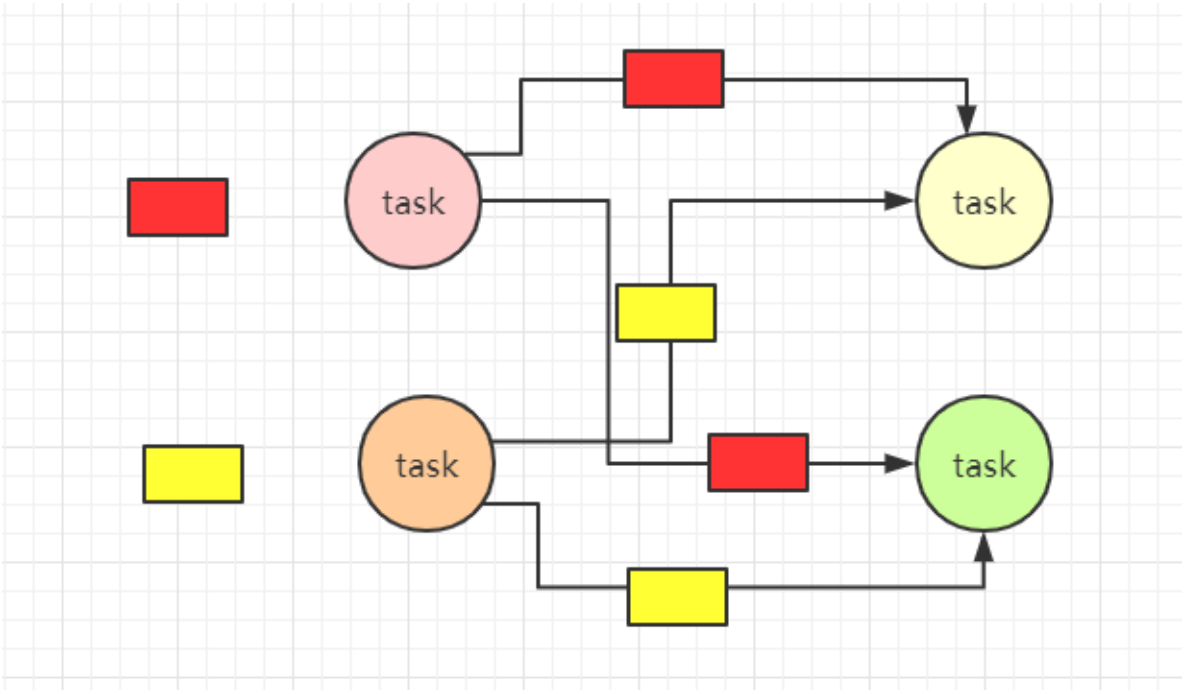
1. 一个 task 的输出只发送给一个 task 作为输入
2. 如果两个 task 都在一个 JVM 中的话，那么就可以避免网络开销

2. key based strategy （keyBy）

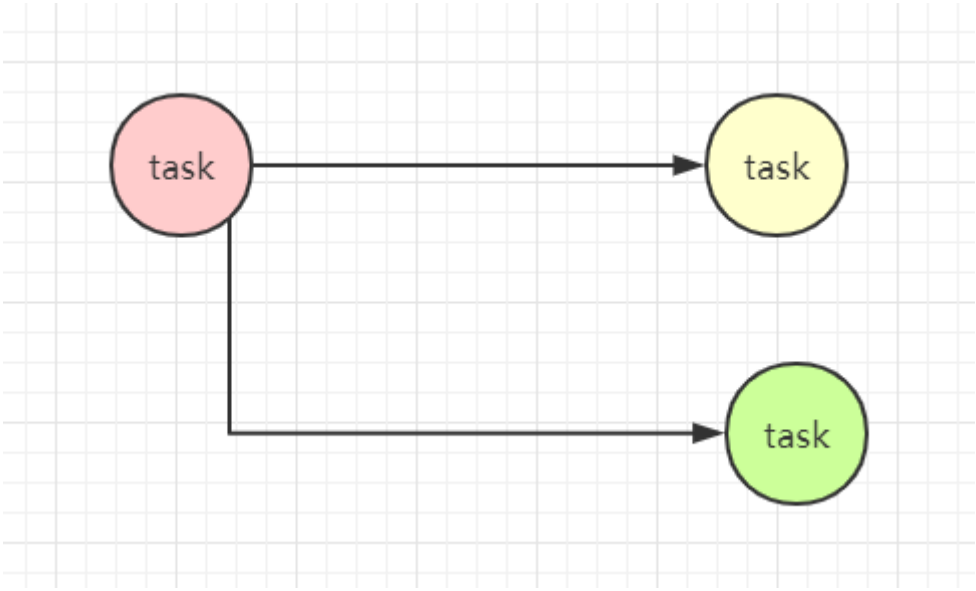


1. 数据需要按照某个属性(我们称为 key)进行分组(或者说分区)
2. 相同 key 的数据需要传输给同一个 task，在一个 task 中进行处理

3. broadcast strategy



4 random strategy



1. 数据随机的从一个 task 中传输给下一个 operator 所有的 subtask
2. 保证数据能均匀的传输给所有的 subtask

TaskManger 并行度 Task Task的数据传输的策略

4.4.5 Operator Chain

1. Flink与Kafka版本整合

Maven Dependency	Supported since	Consumer and Producer Class name	Kafka version	Notes
flink-connector-kafka-0.8_2.11	1.0.0	FlinkKafkaConsumer08 FlinkKafkaProducer08	0.8.x	Uses the SimpleConsumer API of Kafka internally. Offsets are committed to ZK by Flink.
flink-connector-kafka-0.9_2.11	1.0.0	FlinkKafkaConsumer09 FlinkKafkaProducer09	0.9.x	Uses the new Consumer API Kafka.
flink-connector-kafka-0.10_2.11	1.2.0	FlinkKafkaConsumer010 FlinkKafkaProducer010	0.10.x	This connector supports Kafka messages with timestamps both for producing and consuming.
flink-connector-kafka-0.11_2.11	1.4.0	FlinkKafkaConsumer011 FlinkKafkaProducer011	0.11.x	Since 0.11.x Kafka does not support scala 2.10. This connector supports Kafka transactional messaging to provide exactly once semantic for the producer.
flink-connector-kafka_2.11	1.7.0	FlinkKafkaConsumer FlinkKafkaProducer	>= 1.0.0	This universal Kafka connector attempts to track the latest version of the Kafka client. The version of the client it uses may change between Flink releases. Starting with Flink 1.9 release, it uses the Kafka 2.2.0 client. Modern Kafka clients are backwards compatible with broker versions 0.10.0 or later. However for Kafka 0.11.x and 0.10.x versions, we recommend using dedicated flink-connector-kafka-0.11_2.11 and flink-connector-kafka-0.10_2.11 respectively.

添加如下依赖：

```
 <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-connector-kafka_2.11</artifactId>
            <version>${flink.version}</version>
 </dependency>
```

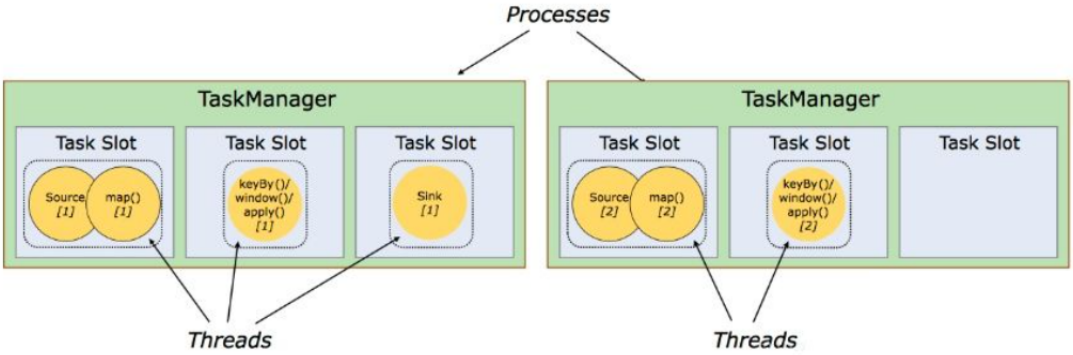
```
public class WordCount {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        String topic="testSlot";
        Properties consumerProperties = new Properties();

 consumerProperties.setProperty("bootstrap.servers","192.168.152.102:9092");
        consumerProperties.setProperty("group.id","testSlot_consumer");


        FlinkKafkaConsumer<String> myConsumer = new FlinkKafkaConsumer<>(topic,
new SimpleStringSchema(), consumerProperties);

        DataStreamSource<String> data =
env.addSource(myConsumer).setParallelism(3);

        SingleOutputStreamOperator<Tuple2<String, Integer>> wordOneStream =
data.flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {
            public void flatMap(String line,
                                Collector<Tuple2<String, Integer>> out) throws
Exception {
                String[] fields = line.split(",");
                for (String word : fields) {
```

```
                    out.collect(Tuple2.of(word, 1));
                }
            }
        }).setParallelism(2);

        SingleOutputStreamOperator<Tuple2<String, Integer>> result =
wordOneStream.keyBy(0).sum(1).setParallelism(2);

        result.map( tuple -> tuple.toString()).setParallelism(2)
                .print().setParallelism(1);

        env.execute("WordCount2");

    }
}
```

2. dataflow效果图如下

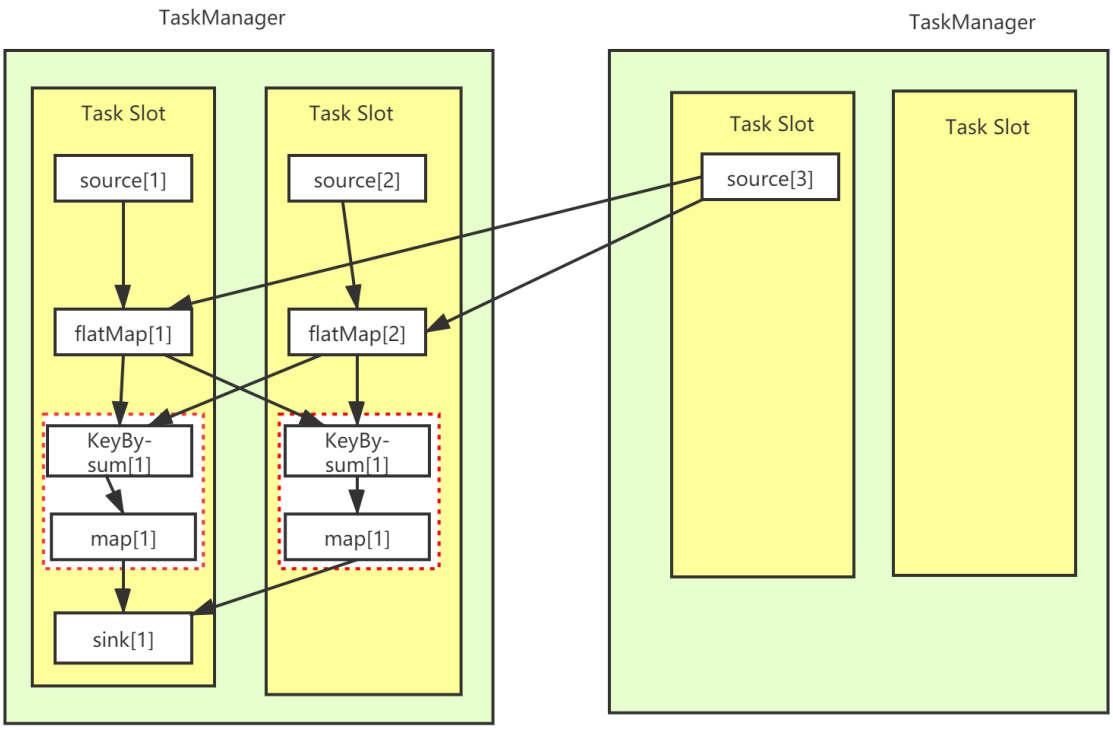
注：演示资源不足，修改资源以后，任务正常运行

3. Task Slot



如何计算我们一个任务里面有几个Task

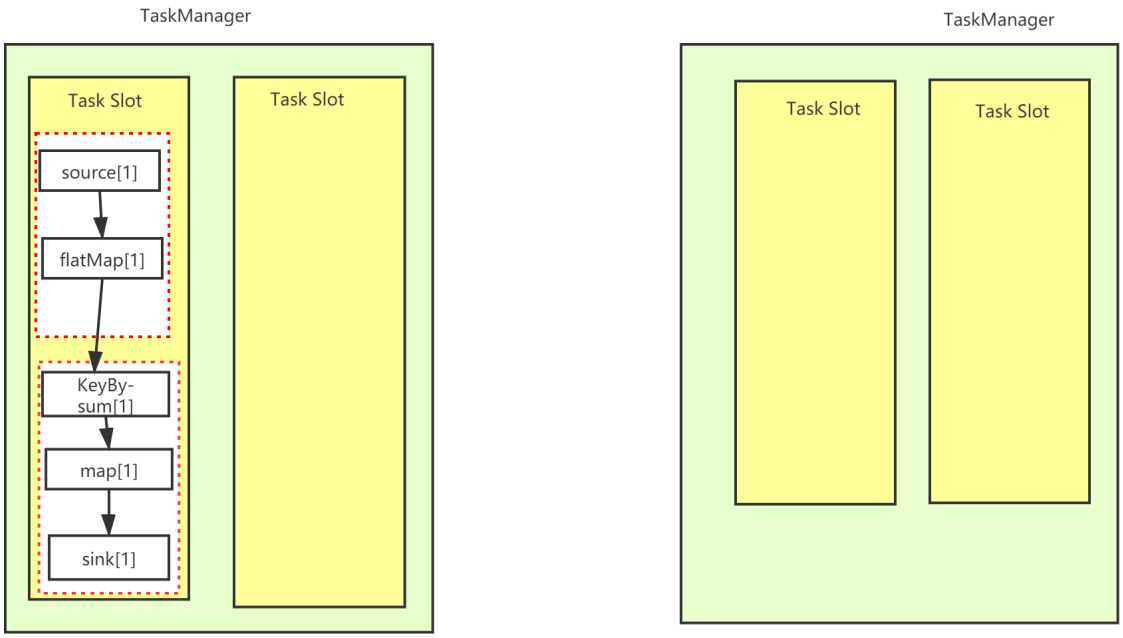
1. Operator Chain



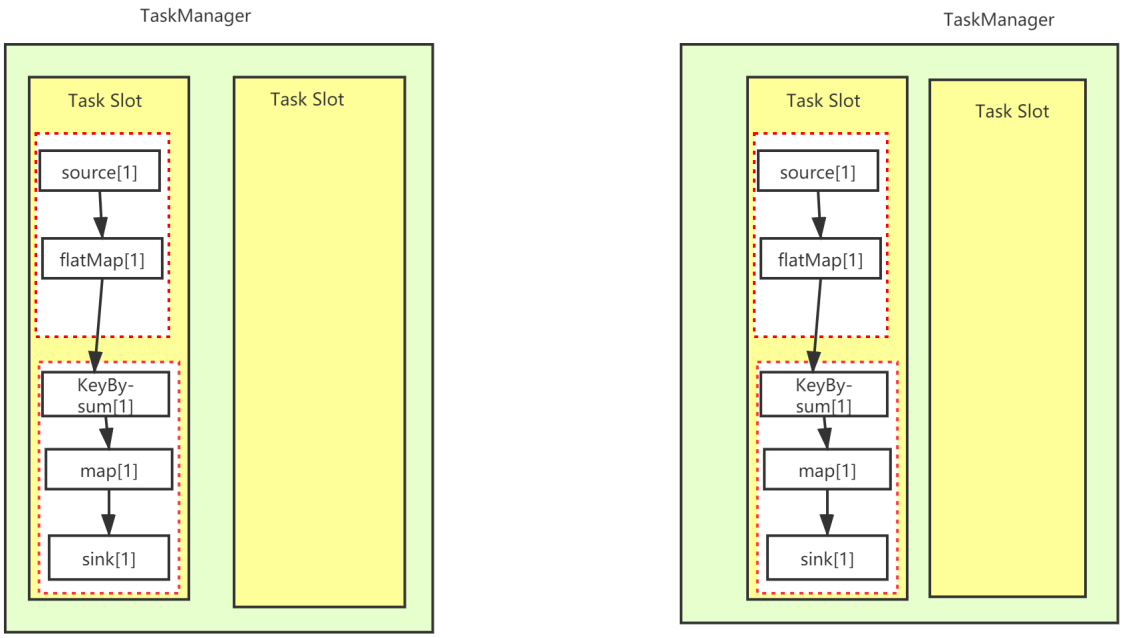
Operator Chain的条件：

1. 数据传输策略是 forward strategy
2. 在同一个 TaskManager 中运行

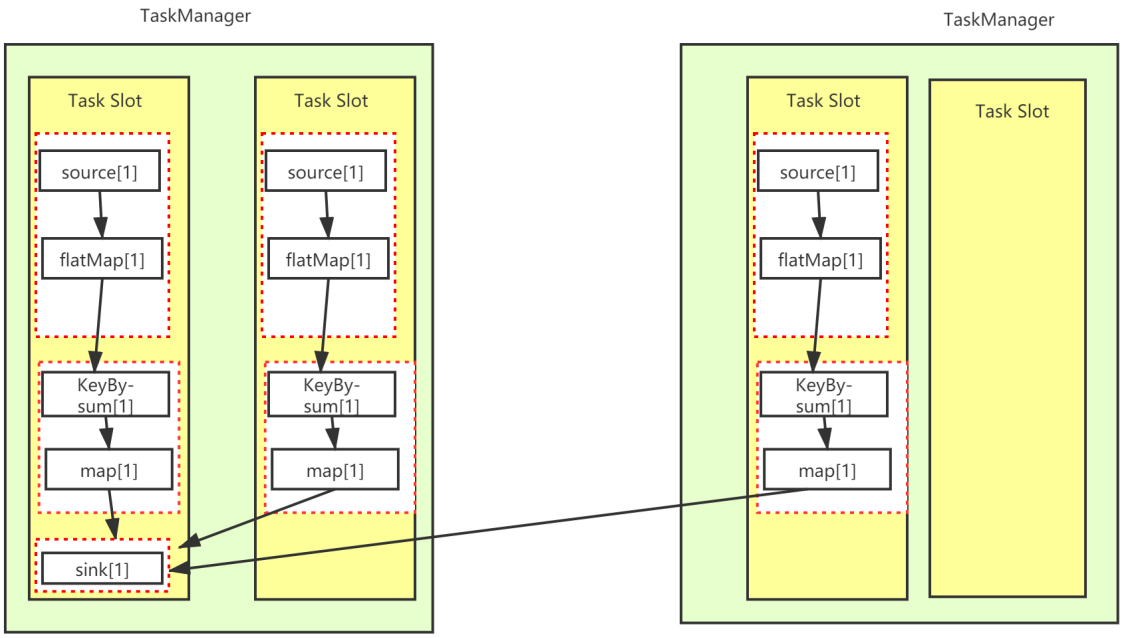
并行度都设置为1，观察情况



并行度设置为2，观察情况



并行度设置为3，观察情况



4.5 Flink分布式运行环境

4.5.1 Flink分布式四层模型

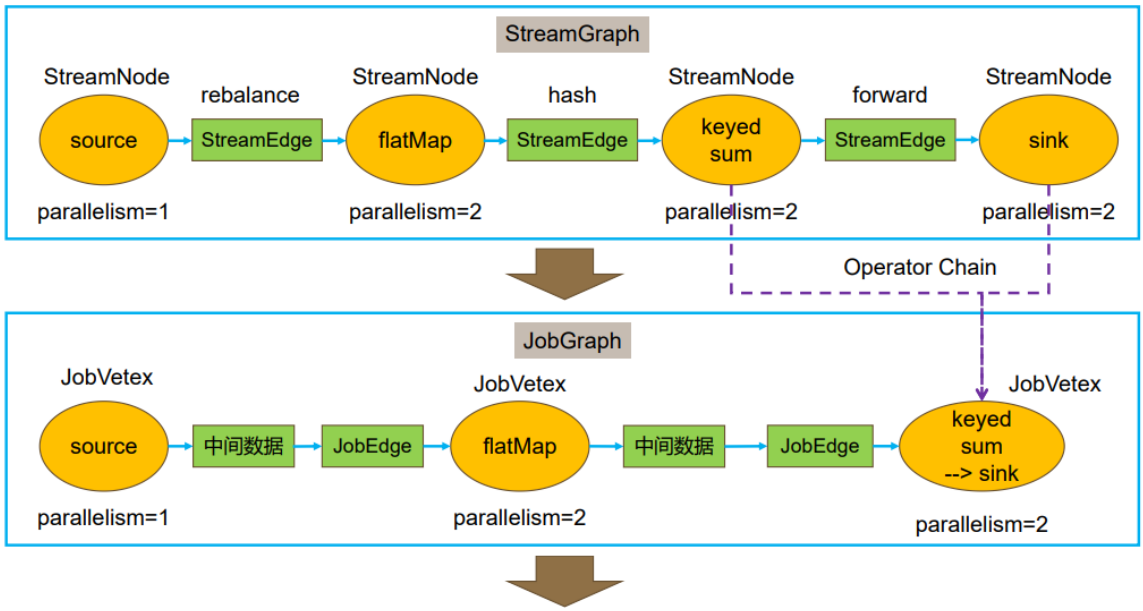
Flink 代码开发就是要构建一个 dataflow，这个 dataflow 运行需要经历如下 4 个阶段：

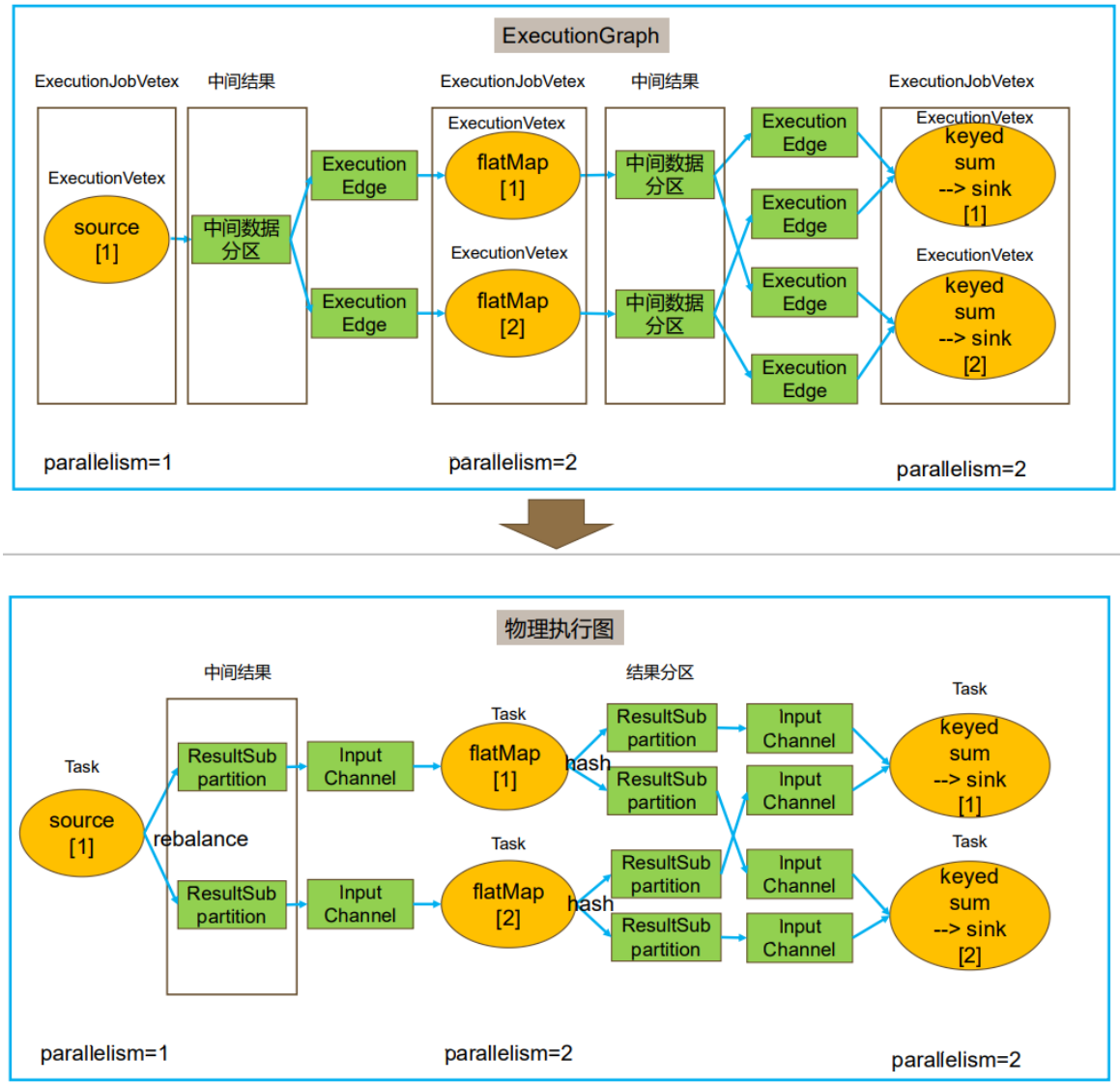
► Stream Graph

► Job Graph

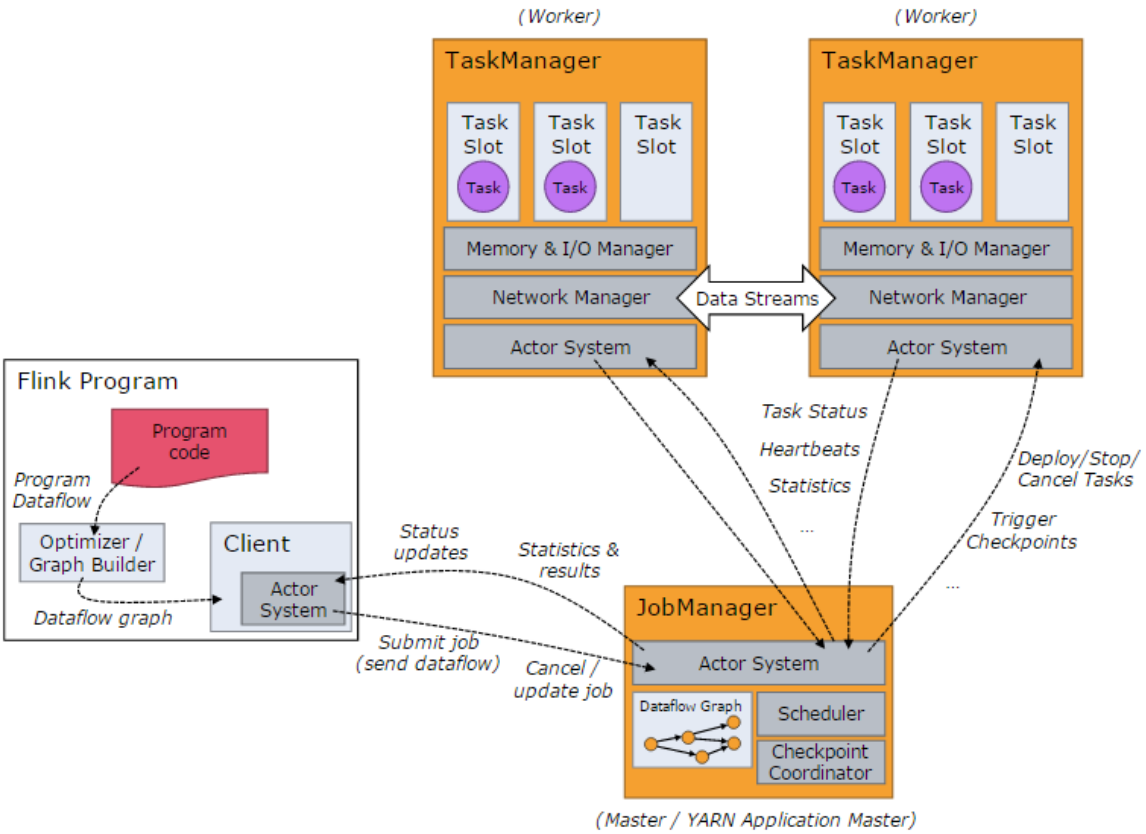
► Execution Graph

► Physical Execution Graph





4.5.2 Flink任务分布式运行流程



五、总结

1. 掌握Flink常见的开发方式
2. 掌握TaskManager，Slot，并行度，Task之间的关系
3. 掌握Flink传输策略
4. 了解Flink分布式四层模型

六、作业